REPORT DOCUMENTATION PAGE

AFRL-SR-BL-TR-00-

0638

1. REPORT DATE *(DD-MM-YYYY)* 9/1/00	2. REPORT TYPE Final	3. DATES COVERED *(From - To)* 6/1/95-5/31/98
4. TITLE AND SUBTITLE Research into Development of Executable Reference Architectures for Avionics Systems		5a. CONTRACT NUMBER F49620-95-1-0378
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) David C. Luckham		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Office of Sponsored Research, Stanford University, Stanford, CA 94305		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFOSR/NI, 110 Duncan Ave, Room B115, Bolling AFB, DC 20332		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

20001127 025

14. ABSTRACT

This project contributed several stufdies and publications on architectures, and avionics architectures, some incollaboration with TRW, Northrop and Lockheed. The references in this and annual reports document this work. This final report presents a summary of a graphics oriented tool called Raparch for developing architecture standards. It has capabilities to design an architecture graphically, input Rapide text to permit an executable form of the architecture for simulation, and planned capaiblities to translate the architecture into other standards languages, including UML and ACME. Rapide and Raparch demos can be invoked on the Rapide Website at Stanford.

15. SUBJECT TERMS

Software design

16. SECURITY CLASSIFICATION OF: Unclassified			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON David Luckham
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified	UL	18	19b. TELEPHONE NUMBER *(include area code)* 650 723 1242

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Evolutionary Development of Complex Systems using Rapide: Transaction Processing Case Study, Fundamental Comcepts, a.k.a Tools Introduction

Introduction

There are several tools to assist programmers who want to develop *Rapide* models of systems. The tools include:

- an architecture-based editor for defining system models,
- a compiler for producing executables from the system models,
- a constraint checking runtime system that is used by an executable to produce a history of the execution,
- a graphical browser for viewing histories, and
- an animation facility providing another view of histories.

This series of documents present an evolutionary, example-driven introduction to these tools. By evolutionary we mean the examples begin with very simple models and progress through increasingly complex models.

All of the models are taken from Dr. Kenney's Ph.D. dissertation on transaction processing. The first models present fundamental concepts of transaction processing. Successive models present the properties of atomicity, isolation and durability. The models conclude with advanced features of security, distribution and performance.

Background

Transaction processing (TP) involves multiple application programs sharing several resources where the application programs make requests of the resources and receive results back. This is a specialization of a client server architecture, where clients make requests of servers. However, TP system architectures have additional properties that distinguish them from client server system architectures. For example, client server architectures do not traditionally have ''transactions,'' groups of requests (and there associated processing) that behave atomically. However, it is required for TP systems to have atomic transactions.

A exempliary TP system architecture contains a single application program that communicates with a set of resources. A "boxes and arrows" depiction of such an architecture is given in Figure 1.

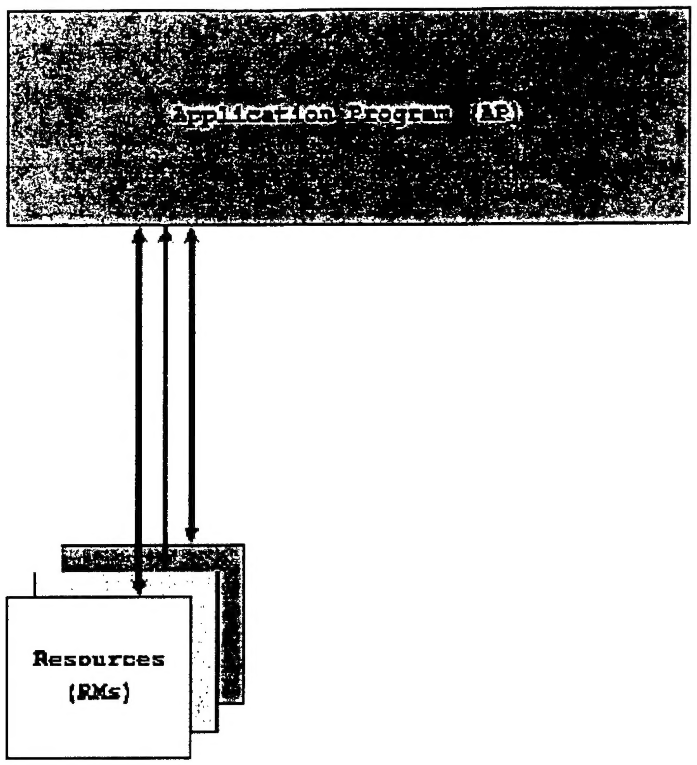


Figure 1: A TP System Architecture.

Figure 1 shows a system consisting of an application program component (at the top) and several (three) resource components (at the bottom). The arrows connecting the components depict that application program may exchange communication with each of the resources, while the resources cannot communicate with each other directly.

We will use the *Rapide* tools to produce evolutionary prototype models of TP. However, to do so we must first install the tools so that you can use them.

Installing *Rapide* Tools

1. Download the appropriate binary file from ftp://pavg.stanford.edu/pub/Rapide-1.0/toolset/. Note: the files are of the form rapide.*type*.build*num*.tar.gz, where *type* is one of:
   - SUNOS - for Sun OS version 4.1,
   - SOLARIS - for Sun OS version 5.5 and 5.6 (aka Solaris 2.5 & 2.6), or
   - LINUX - for Linux version 2.0.

   Generally, the greater the build number *num* the "better" the release. Also, note there are several bug fixes available in the debug subdirectory.
2. Find someplace (preferably /usr) with approximately 50MB of free disk space and extract the tar files:
   - gzip -dc *file* | tar xf -

   (where *file* is the name of the file you downloaded). You should either install the distribution in /usr/rapide, make a symbolic link from /usr/rapide to the distribution, or set the environment variable RAPIDEHOME to the installation directory.
   - ln -s *installation_directory*/rapide /usr/rapide

3. Note: the following instructions assume you have set the RAPIDEHOME environment variable to the installation directory. If you haven't, then either replace the "$RAPIDEHOME" to the *installation_directory*/rapide in the rest of this file or (recommended) set RAPIDEHOME to *installation_directory*/rapide.
   - setenv RAPIDEHOME *installation_directory*/rapide
4. Modify your PATH environment variable to include the rapide "bin" directory:
   - setenv PATH ${RAPIDEHOME}/bin:${PATH}
5. Optionally, modify your MANPATH environment variable to include the rapide "man" directory:
   - setenv MANPATH ${RAPIDEHOME}/man:${MANPATH}
6. Additionally, the installation and execution of the *Rapide* toolset require two environment variable to be properly set. Your path must include a directory that contains the 'make' program, and your shared library search path must include a directory that contains the xview library libxview.so.3.
   - setenv PATH *directory_with_make*:${PATH}
   - setenv LD_LIBRARY_PATH *directory_with_libxview*:${LD_LIBRARY_PATH}
7. Lastly, the Rapide predefined library must be recompiled and a few C++ libraries must be freshened. Please execute the program:
   - $RAPIDEHOME/bin/installation_setup
8. If you have any problems, please refer to the *Rapide* FAQ at:
   - http://pavg.stanford.edu/rapide/FAQ.html

Environment Variables

As discussed in the previous section, the *Rapide* tools generally assumes the *Rapide* distribution is installed under the /usr/rapide directory and that /usr/rapide/bin is included in your path. If you installed the installation in another location, you should set the environment variable RAPIDEHOME to that location and include $RAPIDEHOME/bin in your PATH environment variable. Also, you should appropriately set MANPATH and LD_LIBRARY_PATH.

System Architecture

Normally, the first step in building a prototype of a system is to develop an architecture. The *Rapide* toolsuite provides several ways to do build system architectures, and perhaps the best first step is to use the tool raparch.

Raparch -- The Rapide Architecture Editor

Raparch is a tool for editing Rapide architectures graphically. Boxes and lines may be drawn and edited which corresponds to modules and connections in the architecture of a Rapide program.

Run the raparch program by typing:

```
    % raparch &
```

A box will appear with several menus at the top, two sets of tool bars on the left, and a grid (called a canvas) in the center. Canvas is equipped with a horizontal and a vertical rulers for convenient positioning of objects. The architecture name is specified in the *entry widget* (labelled "Arch Name:") at the bottom. There are two sets of tool bars on the lefthand side of a canvas. The first group is for drawing and editing components and connections, such as ▬ (create a component) or ╹ (create a connection) mode. The second group of tool bars is used to depict features of a component for its behavior specifications. A tool bar in this group can only be activated in conjuction with ↗ (select) mode of the first tool bar group.

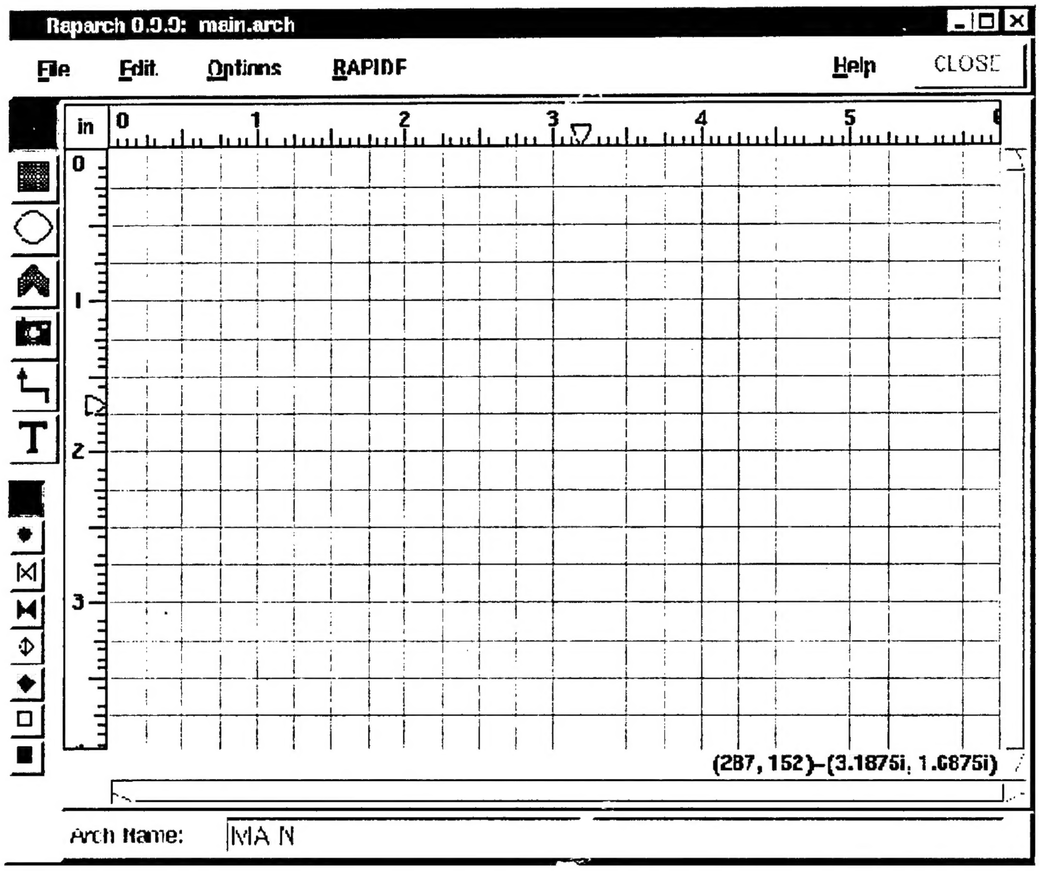


Figure 2: Initial View of Raparch

System Architecture (Picture)

The next step is to decide on an intial system architecture for your model. In this example we use a single application program connected to a single resource. This architecture is depicted in Figure 3, and we will now use raparch to construct a model of this architecture.

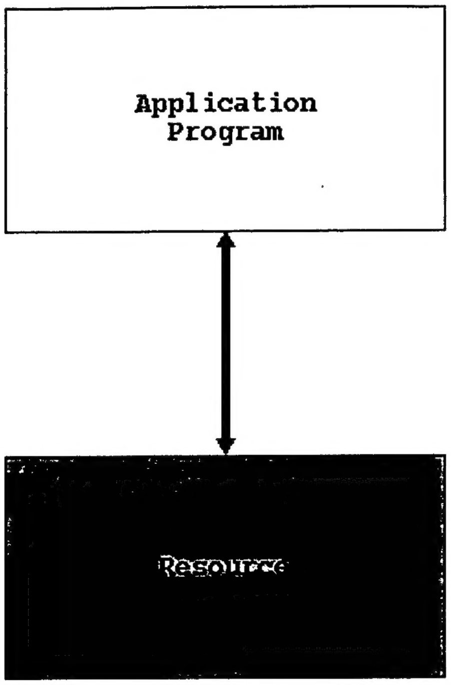


Figure 3: Single Application Program and Resource System Architecture.

Create an Application Program Component

To create an application program component, first enter the "create rectangular modules" mode by clicking on the ▨ icon in the first tool bar group. Then create a box towards the top of the canvas for the application program by positioning the cursor towards the top of the canvas where you want the top left hand corner of the box to be. Then press down, drag and then release left mouse button (or LeftButton) where you want the bottom right hand corner of the box to be. You will see a box labelled "Module 1" appear.

Modify the module properties of application program component by (1) being in ▨ mode, and (2) pressing down in the "Module 1" box on <Alt>-RightButton. An associated module properties window will appear. Modify the "Module Name" to be "APPLICATION_PROGRAM" and modify the "Module Label" to be "Application\nProgram." (The "\n" in the label means a newline) Then click on the OK button.

Create a Resource Component

Similarly, create another component at the bottom of the canvas for the resource by: (1) being in ▨ mode, and (2) press down, drag and release the LeftButton from the top left hand corner to the bottom right hand corner.

Modify the modules properties of the resource component by (1) being in ▨ mode, and (2) pressing down in the "Module 2" box on <Alt>-RightButton. Modify the "Module Name" to be "RESOURCE" and modify the "Module Label" to be "Resource." Optionally, you may change the color of the module by typing in a color or using the "Choose..." button to select a color. Finally, click on OK.

Create a Path between the Components

To create a path between the application program and resource components, first enter the "connect modules" mode by clicking on the button in tool bar with the ↰ icon. Press and release the <Shift>-LeftButton on the application program component and then again on the resource component. This will produce a double headed arrow between the components.

A double headed arrow creates a ''two-way'' connection between the components; by two-way we mean that in the animation that you will produce later in the this tutorial, events will traverse in both directions along this pathway. If you desire a ''one-way'' connection, which limits events to tranverse along the pathway in only the direction of the arrow, then press and release the LeftButton on the source component and <Shift>-LeftButton on the sink component.

Note: A pathway without either arrow heads defaults to one-way in the direction the pathway was created. The direction of the arrow can be changed using a path properties dialogue box.

Modify the connection's path properties by pressing and releasing <Alt>-RightButton anywhere along the path while you are in the ↰ mode. For example, change the "Path Width" to 3. Finally, click on OK.

Optionally, you can modify the direction of arrow and connection mode by clicking the appropriate radio button for the corresponding field. Like in a component properties window, you may change the color of the path by typing in a color or using the "Choose..." button to select a color. A path name can optinally be specified by typing in a particular name for the path as well.

Resizing and Repositioning

Optionally, you can move a component when you are in the "Selection tool" mode by pressing down on the ↗ icon, pressing down and dragging the component with LeftButton. Notice that when a component is selected it turns blue.

Optionally, you can resize a component when you are in ▨ mode by pressing down and dragging an edge of the box with <Shift>-RightButton.

Optionally, you can reposition a path when you are in ↰ mode by pressing down and dragging an end of the path with <Shift>-RightButton.

Saving the Architecture

The architecture picture that we have built can be saved two ways. First, saved to a Rapide source file that can be compiled and executed (saved in a file with ".rpd" extension by default). Second, saved to an architecture file that can be used to animate the results of an execution (saved in a file with ".arch" extension by default).To perform the latter, select the Save option of the File menu. This creates a file named "main.arch" that contains a description of the architecture in a format that is understood by raparch and raptor. You can save the description in file with another name by using the Save As option of the File menu.

Note: If you save the file under a different name, be sure to use that name instead of main.arch throughout the rest of the example! So far we are building a conceptual (initial) model of the system architecture. We save this conceptual architecture in "main_concept.arch" by the Save As option of the File menu. The file name is inherited from the name used in "Arch Name" field at the bottom of the canvas. A default architecture name used is "main".

System Behavior

Rapide (and raparch) allows us to associate prototypical behaviors to our system. We can provide the architecture with sementics, in other words, we are constructing a *structural* and *behavioral* model of the system architecture.

Application Program Behavior

To associate a behavior with the application program component, first we create required features for the component: (1) being in ↗ mode in the first tool bar group, (2) clicking on the ○ icon in the second tool bar group. Then create an *out action* feature over the boundary of application program component by <Control>-MiddleButton (middle mouse button on 3-button mouse or both left and right mouse button pressed simultaneously on 2-button mouse).

Modify the properties of out action feature of application program component by (1) being in ↗ mode , (2) being in ○ mode , and (3) pressing down in the "out action" feature on <Alt>-RightButton. Modify the "Name" to be "Request". Similarily we create the other *in action* feature for application program component and name it to be "Result". Now, the structural interface of application program component is established and the next step is to specify a behavior for application program component based on its constituting *in* and *out action* features.

To describe a behavior for the application program component, you must be in ■ mode and press <Control>-RightButton on the application program component. This brings up a window that defines the *Rapide* interface type associated with the application program component. At this point, raparch should provide the structural interface constituents in "acitons and services" section of the type interface window properly. (note: sometimes pressing <Control>-RightButton over components causes the architecture description window open. This is a minor raparch bug to be fixed in the next release. If that happens, ignore the architecture window by clicking the Cancel Button in the window.) Make sure that actions and serives section contains:

```
action out  Request();
action in   Result();
```

These action declarations mean that the application program can generate "out" `Request` events and receive "in" `Result` events. These actions will be "visible" to the resource component as we shall see later.

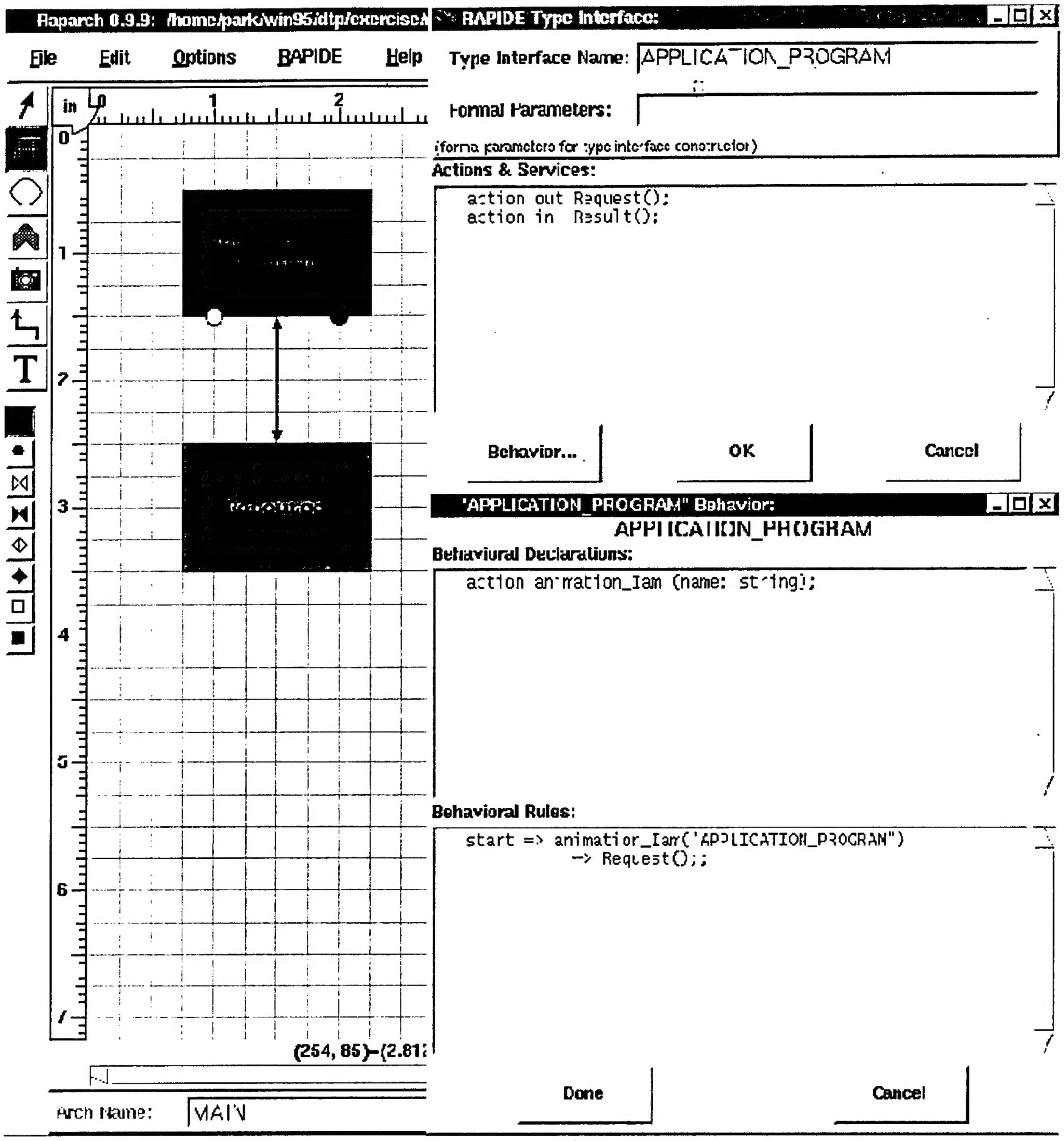
Click on the "Behavior..." button to describe the component's behavior rules. It will bring up a new window for the component behavior, labelled "APPLICATION_PROGRAM." There are "Behavioral Declarations" and "Behavioral Rules" sections in the new window. Interfaces in *Rapide* can also include behavioral declarations that we will initially leave alone. We will modify them later to enrich our behavioral specification. Notice the default action declaration that is special. This action's associated events communicate to raptor the "name" of the component that generated the event.

Modify the behavioral rules section to be:

```
start =>
  animation_Iam("APPLICATION_PROGRAM")
   -> Request();;
```

This behavior rule waits until a start event is observed and reacts to it by generating two events: an animation_Iam event and a `Request` event. The animation_Iam event is parameterized with the string "APPLICATION_PROGRAM."

When you are finished making the modifications, click on the DONE button for behavioral rules window and OK button for type interface window. The snapshot of this architecting process is depicted in Figure 4.



Type Interface Name: APPLICATION_PROGRAM

Formal Parameters:

(formal parameters for type interface constructor)

Actions & Services:

```
  action out Request();
  action in  Result();
```

APPLICATION_PROGRAM

Behavioral Declarations:

```
  action animation_Iam (name: string);
```

Behavioral Rules:

```
  start => animation_Iam('APPLICATION_PROGRAM")
           -> Request();;
```

Arch Name: MAIN

Figure 4: Application Program Component Features and its Behavioral Descriptions.

Resource Behavior

Similarily now we will associate a behavior with the resource component. First, we create required features for the resource component, as explained in the above: (1) being in ↗ mode , (2) clicking on the ○ icon , then create an *out action* ("Result") and an *in action* ("Request"). The structural interface of resource component should be properly collected by raparch and be shown in type interface window. Associated behavior can be specified in the same way as we described for the applicaiton program component. Make sure that resource componet contains:

```
action  in  Request();

action  out Result();
```

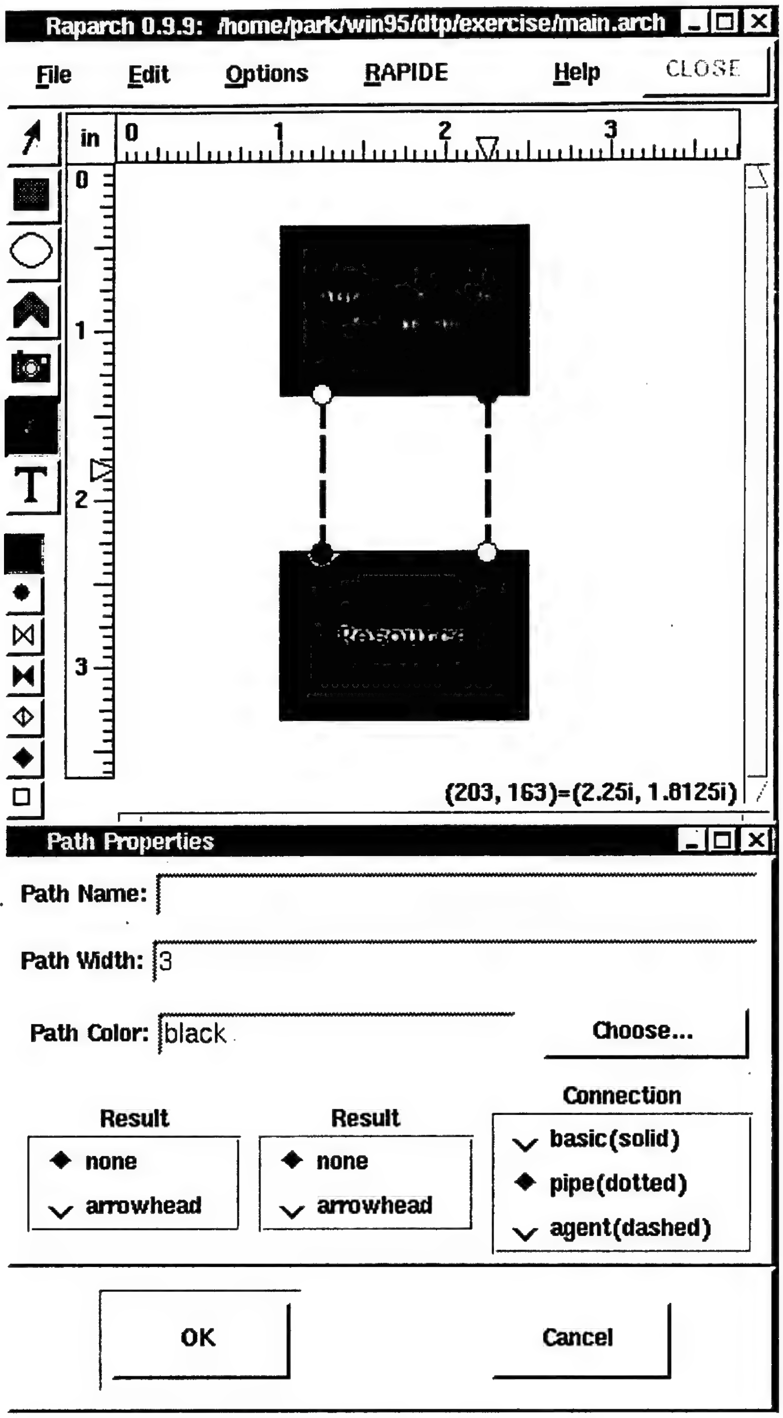
And modify the "behavioral rules" section in a behavior window to be:

```
start => animation_Iam("RESOURCE");;
Request() => Result();;
```

When you are finished making the modifications, click on the DONE button for behavioral rules window and OK button for type interface window. The snapshot for RESOURCE component behavior modification can be shown in a similar figure like Figure 4.

Create Pathes between the Features of Components

Modify connections for the system. First, delete a connection made between application program and resource components by selecting the path between two components and using Delete option of the Edit menu. Instead we will make a connection between two components, in terms of components' features, that is, "Request" and "Result" action pair of each component. To create a path between features of components, first enter the "connect modules" mode by clicking on the button in the first tool bar group with ↰ icon. Press and release LeftButton on the *out action* "Request of Application Program" and then again on the *in action* "Result of Resource" component.. This will produce a solid line between two features. Modify the connection's path properties by pressing and releasing <Alt>-RightButton anywhere along the path while you are in the ↰ mode. For example, modify the "Path Width" to 3 and connection mode to be "pipe" for now, then click on the OK button. You will see a change of line pattern for a path, from a solid line to a dotted line. Details about connections will be discussed later in this tutorial. Note: Since features of components already implies the direction of flow (e.g., *out* or *in*), arrows may not be required for connections between features. A solid line represents a *basic* connection and a dotted line for a *pipe* connection in Rapide. Figure 5 shows two *pipe* connections between Request and Result action pairs of ApplicationProgram and Resource components in the architecture.



Raparch 0.9.9: /home/park/win95/dtp/exercise/main.arch

(203, 163)=(2.25i, 1.8125i)

Path Properties

Path Name:

Path Width: 3

Path Color: black

Result: [✓ none] arrowhead

Result: [✓ none] arrowhead

Connection: basic(solid) [✓ pipe(dotted)] agent(dashed)

Figure 5: Connections between Components based on their Features.

Main Architecture

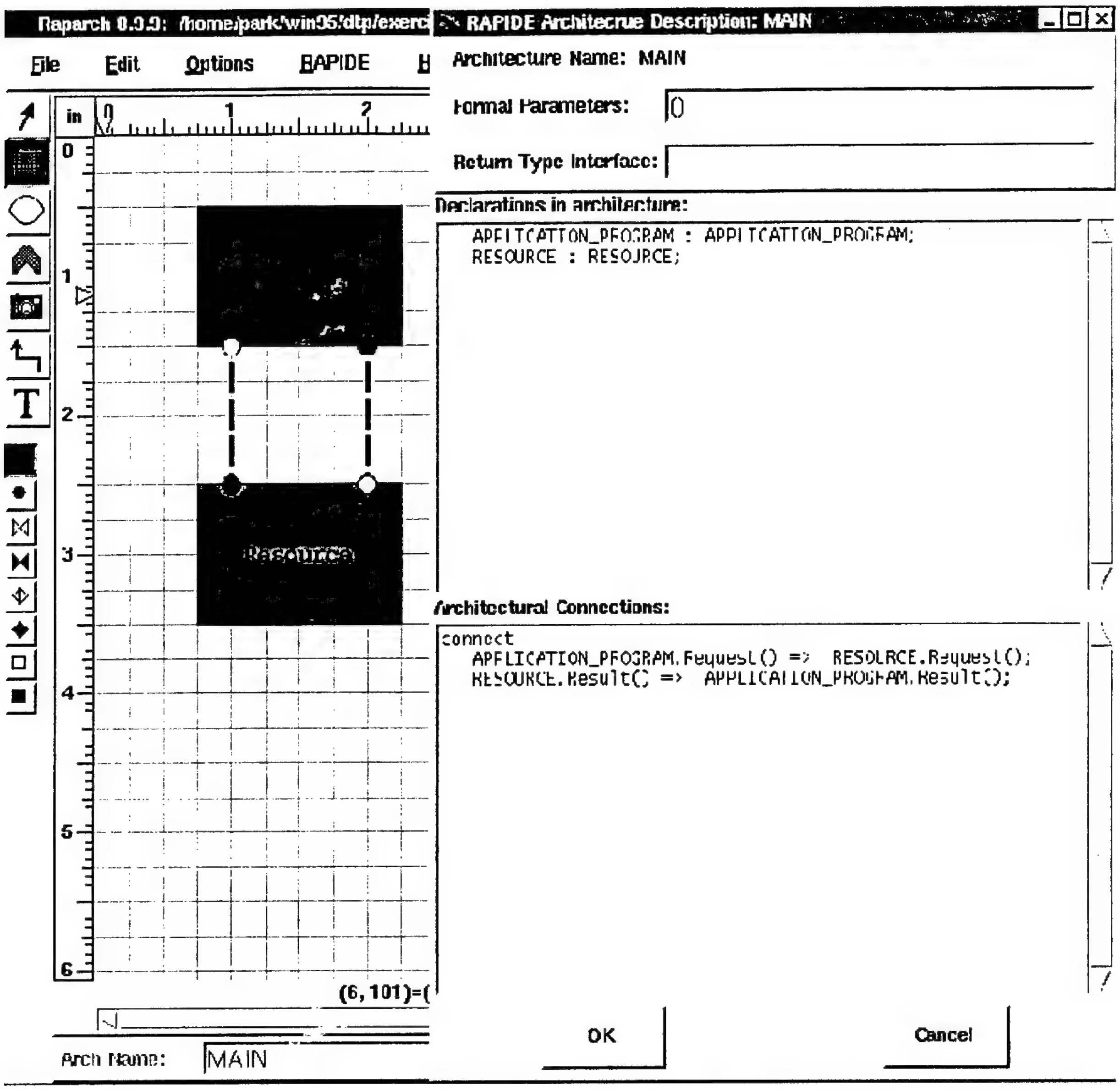
Now we will associate code with the "main" architecture. In either ■ or ↗ mode and press <Control>-RightButton on an empty portion of the canvas to bring up the *Rapide* architecture description window. Once agiain raparch should be able to collect constituting components and connections automatically. Make sure that main architecture window contains the following declarations in "Decalarations in architecture" section. Note: Current raparch implementation may need a bit of massages for *Rapide* source code generation in some cases. "Declarations in architecture" section shows APPLICATION_PROGRAM and RESOURCE declarations separated by ":". Names prior to ":" represent components' identifiers in the architecture, and following APPLICATION_PROGRAM and RESOURCE represents *Rapide* interface types.

```
APPLICATION_PROGRAM : APPLICATION_PROGRAM;
RESOURCE   : RESOURCE;
```

And make sure "Architectural Connections" section to be:

```
connect
   APPLICATION_PROGRAM.Request() => RESOURCE.Request();
   RESOURCE.Result() => APPLICATION_PROGRAM.Result();
```

When you are finished making the modifications, if needed, and click on the OK button. The situation is depicted in Figure 6.


Raparch 0.3.9: /home/park/win95/dtp/exerc
RAPIDE Architecrue Description: MAIN
File
Edit
Options
RAPIDE
in
Resource
(6,101)=(
OK
Cancel

Architecture Name: MAIN

Formal Parameters: ()

Return Type Interface:

Declarations in architecture:

```
   APPLICATION_PROGRAM : APPLICATION_PROGRAM;
   RESOURCE : RESOURCE;
```

Architectural Connections:

```
connect
   APPLICATION_PROGRAM.Request() =>  RESOURCE.Request();
   RESOURCE.Result() =>  APPLICATION_PROGRAM.Result();
```

Arch Name: MAIN

Figure 6: Main Architecture Description

Generating the *Rapide* code

The architecture code that you have specified can be used to generate a *Rapide* program via the Generate Rapide Code option of the RAPIDE menu. Enter "main.rpd" in the selection widget to create a file named main.rpd.

When you are finished making the selection, click on the OK button. You will have produced a file named main.rpd containing the following *Rapide* program.

```
TYPE APPLICATION_PROGRAM IS INTERFACE
    action out  Request();
```

```
        action in   Result();
      BEHAVIOR
        action animation_Iam(name: string);
      BEGIN
        start => animation_Iam("APPLICATION_PROGRAM") -> Request();;
    END;

    TYPE RESOURCE IS INTERFACE
        action in  Request();
        action out Result();
      BEHAVIOR
        action animation_Iam(name: string);
      BEGIN
        start => animation_Iam("RESOURCE");;
        Request() => Result();;
    END;

    ARCHITECTURE MAIN() IS
        APPLICATION_PROGRAM : APPLICATION_PROGRAM;
        RESOURCE   : RESOURCE;
      CONNECT
        APPLICATION_PROGRAM.Request() => RESOURCE.Request();
        RESOURCE.Result() => APPLICATION_PROGRAM.Result();
    END;
```

Also, save raparch's state with the Save option of the File menu. This completes the construction of the structural and behavioral model, in two files named "main.arch" and "main.rpd."

System Execution

The code saved in main.rpd can be compiled, executed and analyzed.

R.manager -- The *Rapide* Library Manager

To compile a *Rapide* program, a *Rapide* library must first be built. A *Rapide* library store information about the predefined and user defined types and modules. To create a library, type:

```
% r.mklib
```

Rpdc -- The *Rapide* Compiler

To compile the code in main.rpd to create an executable type:

```
% rpdc -M main -o main main.rpd
```

Compiling the program should take approximately 1 minute on a Sun Ultra. If you get errors during the compilation, please check the associated sections of the code for typographical errors. Upon correcting the error, don't forget to save the program.

Execute the program by typing:

```
% main
```

This will produce a history of the execution recorded in main.log.

System Analysis

There are several forms of analysis provided by the *Rapide* toolsuite. In particular, we will use raptor and pov.

Raptor -- The *Rapide* Animator

To create a graphical animation of what happened during the execution, type:

```
    % raptor -a main.arch main.log &
```

Pov -- The *Rapide* Partial Order (poset) Viewer

We can visualize what happened during the execution another way using the pov. The pov can create several views of the events that were generated during the execution. In particular, a directed acyclic graph can be generated whose arrows denote the causal (or more accurately the dependency) relationship between the events. If an arrow goes from an event A to another event B then event B depends on event A.

The pov is invoked with the log file name as its command line argument:

```
    % pov main.log &
```

The pov starts initially with the view manager window which lists the computations (and views) that have been loaded; at this point only one computation has been loaded, the main.log, and it has been given the label "Computation 1." To examine the poset graphically, select Computation 1 and use the poset viewer option of the View menu. Also, try the table viewer to represent the events in a tabular form.

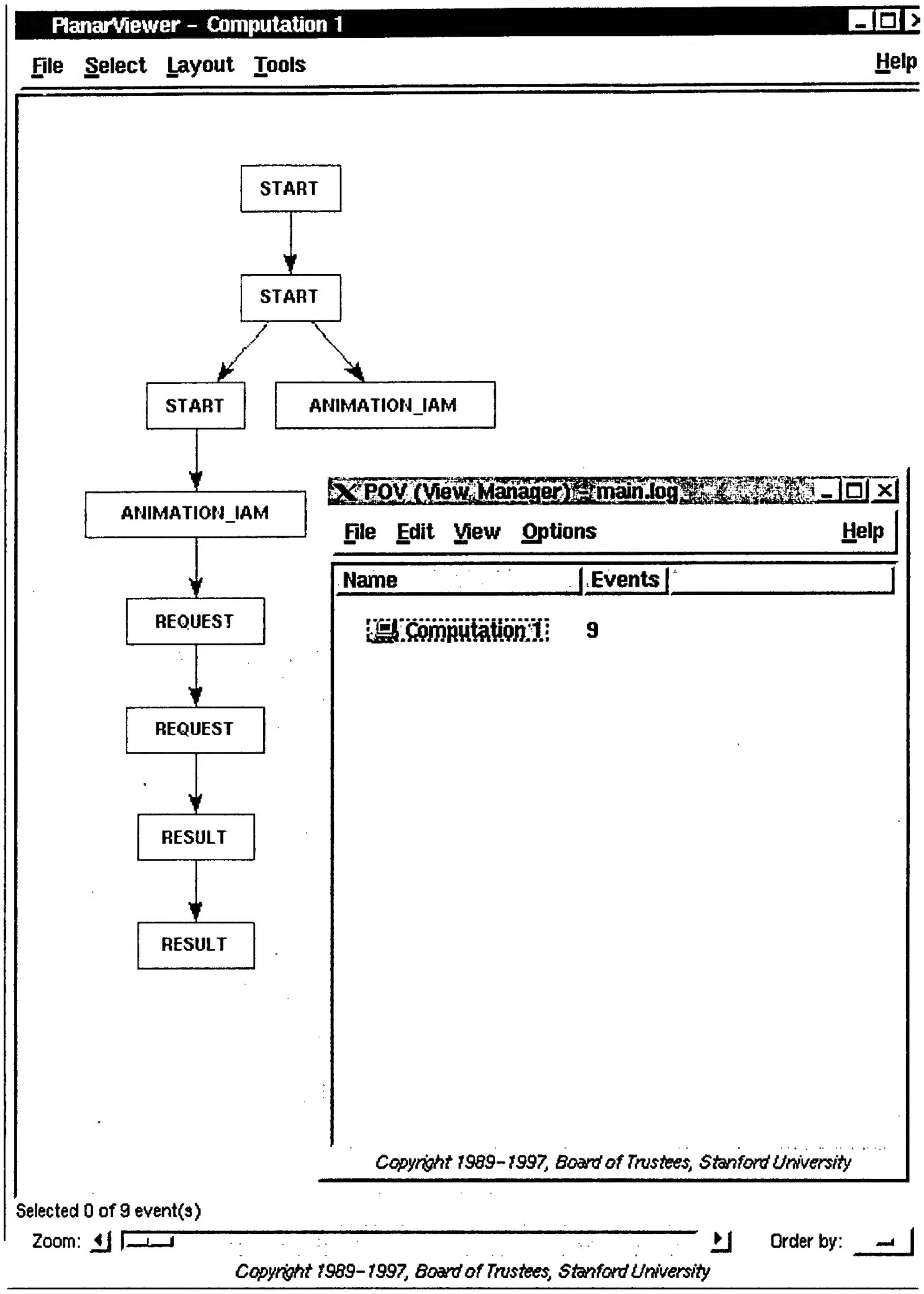

PlanarViewer - Computation 1
File Select Layout Tools Help
START
START
START
ANIMATION_IAM
ANIMATION_IAM
REQUEST
REQUEST
RESULT
RESULT
POV (View Manager) main.log
File Edit View Options Help
Name
Events
Computation 1
9
Copyright 1989-1997, Board of Trustees, Stanford University
Selected 0 of 9 event(s)
Zoom:
Order by:
Copyright 1989-1997, Board of Trustees, Stanford University

Figure 6: Pov's PlanarViewer of main Execution.

If you are running a secure X server, you can coordinate the animation's displaying of the events with modification of the pov's poset viewer's coloring of the events. To coordinate the animation with the visualization, you may drag and drop the computation from the pov to raptor. One way to do this is to click and hold RightButton on "Computation 1" in the view manager window of the pov and drag the computation over to the raptor window. When the dragged computation turns green, you may release the button. If the computation doesn’t turn green, you are probably not running a secure X server.

REFERENCES:

1. NIST 4D/RCS Reference Model Architecture:
www.itl.nist.gov/div97/projects/projad1.htm

2. B2 Avionics Modeling and Analysis with Rapide (JSF)
www.northgum.com/ng_review/review04_03.html

3. MISSI Reference Architecture -- From Prose to Precise Specification
by Sigurd Meldal and David C. Luckham

4. Towards and Abstraction Hierarchy for CAETI Architectures
by D. C. Luckham, F. Belz and J. Vera
CSL Stanford University, 1997 CSL-TR-97-727

5. Complex Event Processing in Distributed Systems
by D. C. Luckham and B. Frasca
CSL Stanford University, 1998 CSL-TR-98-754